p5-search-engine

EECS 485 Project 5: Wikipedia Search Engine

Project due 12/10/2018

Current Version 3.0-f18

Build a scalable search engine that is similar to a commercial search engine. The search engine in this assignment has several features:

- Indexing implemented with MapReduce so it can scale to very large corpus sizes
- Information retrieval based on both tf-idf and PageRank scores
- A new search engine interface with two special features: user-driven scoring and summarization.

The learning goals of this project are information retrieval concepts like PageRank and tf-idf, parallel data processing with MapReduce, and writing an end-to-end search engine.

IMPORTANT: Please do not commit any large files to your GitHub repo. This includes things like vagrant folders, the Wikipedia input or your inverted index. It's 100s of MBs and will slow down your pushes and pulls, but moreover, will make syncing with your partners painful. Update your `.gitignore` ASAP!

Since your large input files won't be on Github, use the scp command line program to copy them over to the server.

You can find the starter files in the starter_files folder.

Table of Contents

Part 0: Setup

We'll use a similar environment for this project that we used for past projects. Refer to the P1 setup tutorial for setting up your development environment.

Download the starter files and copy everything into $PROJECT_ROOT. This is what your directory should look like then.

```
.
├── VERSION
├── bin
│   ├── hadoop_pipeline
│   └── submit
├── index
│   ├── hadoop
│   │   ├── exec
│   │   │   ├── example
│   │   │   │   ├── map.py
│   │   │   │   └── reduce.py
│   │   │   └── inverted_index
│   │   │       ├── map0.py
│   │   │       └── reduce0.py
│   │   ├── hadoop-streaming-2.7.2.jar
│   │   ├── mapreduce_input_data
│   │   ├── sample.txt
│   │   └── sampleInput
│   │       ├── file01
│   │       └── file02
│   ├── index
│   │   ├── pagerank.out
│   │   └── stopwords.txt
│   └── setup.py
├── search
│   ├── package.json
│   ├── package-lock.json
│   ├── search
│   │   └── sql
│   │       └── wikipedia.sql
│   ├── setup.py
│   └── webpack.config.js
└── stopwords.txt
```

This project involves very few starter files. You have learned everything you need to build this project from (almost) scratch. You are responsible for converting the starter files structure into the final structure. Each part of the spec will walk you through what we expect in terms of structure!

- `bin/` : As in previous projects, this is where convenience shell scripts will be located. You will write "search", "index", "db", and "hadoop_pipeline".

- `index/hadoop` : This is where all hadoop related files will live. See Part 1 and the following Hadoop setup and example instructions for more details.
- `index/index` : This folder will have all of the Python code for the index server. See Part 2 for more details.
- `search/search` : This folder should have your search interface app. See Part 3 for more details.

Both the index server and the search server are separate Flask apps. The search server will serve a bundle.js of your react source code. The index server will be styled after the project 3 RESTful API

As in previous projects, both will be python packages that should have a `setup.py` .

At the end of this project your directory structure should look something like this:

```
.
├── VERSION
├── bin
│   ├── hadoop_pipeline
│   ├── servers
│   └── submit
|── deployed_index.json
├── deployed_index.log
├── deployed_search.html
├── deployed_search.log
├── index
│   ├── hadoop
│   │   ├── exec
│   │   │   └── inverted_index
│   │   │       ├── map0.py
│   │   │       ├── mapX.py
│   │   │       ├── reduce0.py
│   │   │       └── reduceX.py
│   │   ├── hadoop-streaming-2.7.2.jar
│   │   ├── mapreduce_input_data
│   │   └── split_file.py
│   ├── index
│   │   ├── *.py
│   │   ├── api
│   │   │   ├── *.py
│   │   ├── inverted_index.txt
│   │   ├── pagerank.out
│   │   └── stopwords.txt
│   └── setup.py
├── search
│   ├── package.json
│   ├── package-lock.json
│   ├── node_modules
│   │   └── */
│   ├── search
│   │   └── api
│   │   ├── api
│   │   │   └── *.py
│   │   ├── js
│   │   │   └── *.jsx
│   │   ├── server_config.py
│   │   ├── sql
│   │   │   └── wikipedia.sql
│   │   ├── static
│   │   │   └── js
│   │   │       └── bundle.js
│   │   ├── templates
│   │   │   └── *.html
│   │   ├── var
│   │   │   └── wikipedia.sqlite3
│   │   └── views
│   │       └── *.py
│   ├── setup.py
│   └── webpack.config.js
└── stopwords.txt
```

Installing Hadoop

If you are on OSX, install Hadoop with Homebrew

```
$ brew cask install java
$ brew install hadoop
```

If you are on Ubuntu Linux, Ubuntu Linux VM or Windows 10 WSL, you will have to install Hadoop manually. **OSX users skip the following. Resume below, where specified**

Install Java

```
$ sudo -s                             # Become root
$ sudo apt-get update
$ sudo apt-get install default-jdk    # Install Java
$ java -version                       # Check Java version
openjdk version "1.8.0_151"
OpenJDK Runtime Environment (build 1.8.0_151-8u151-b12-0ubuntu0.16.04.2-b12)
OpenJDK 64-Bit Server VM (build 25.151-b12, mixed mode)
```

Download Hadoop and unpack.

```
$ cd /opt
$ wget http://apache.osuosl.org/hadoop/common/hadoop-2.8.5/hadoop-2.8.5.tar.gz
$ wget https://dist.apache.org/repos/dist/release/hadoop/common/hadoop-2.8.5/hadoop-2.8.5.tar
$ shasum -a 256 hadoop-2.8.5.tar.gz
```

```
e8bf9a53337b1dca3b152b0a5b5e277dc734e76520543e525c301a050bb27eae  hadoop-2.8.5.tar.gz
$ grep SHA256 hadoop-2.8.5.tar.gz.mds
SHA256 = E8BF9A53 337B1DCA 3B152B0A 5B5E277D C734E765 20543E52 5C301A05 0BB27EAE
$ tar -xvzf hadoop-2.8.5.tar.gz
$ rm hadoop-2.8.5.tar.gz hadoop-2.8.5.tar.gz.mds
```

Locate the path to your Java interpreter.

```
$ which java | xargs readlink -f | sed 's:bin/java::'
/usr/lib/jvm/java-8-openjdk-amd64/jre/
```

Edit `/opt/hadoop-2.8.5/etc/hadoop/hadoop-env.sh` to change the `JAVA_HOME`.

```
#export JAVA_HOME=${JAVA_HOME}  # remove/comment
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64/jre
```

Write a launch script, `/usr/local/bin/hadoop`

```
#!/bin/bash
exec "/opt/hadoop-2.8.5/bin/hadoop" "$@"
```

Make the script executable and check that it's working.

```
$ chmod +x /usr/local/bin/hadoop
$ which hadoop
/usr/local/bin/hadoop
$ hadoop  -h
Usage: hadoop [--config confdir] [COMMAND | CLASSNAME]
...
$ exit  # drop root privileges
```

OSX users resume here

Note to OSX users: If you run Hadoop and Hadoop is unable to find JAVA_HOME, please refer to the fix in OSX Java Home Error FAQ

"Hello World" With Hadoop

Let's run a toy example MapReduce (MR) using Hadoop. We have provided a sample mapper and reducer executable in `index/hadoop/exec/example` inside the starter files. We have also provided a shell script that will execute this MR job. This is `bin/hadoop_pipeline` (this will eventually be where you run your full MR pipeline from - see Part 1). Right now, hadoop_pipeline is configured to run a one stage pipeline. Run `./bin/hadoop_pipeline` from $PROJECT_ROOT. You will see a `index/hadoop/output` directory created. This is where the output of the MR job lives. You are interested in all “part-XXXXX” files. The SUCCESS files are indicator files of a sucessful MR job. The hidden files (starting with a ‘.’) are extra binary files for hadoop to use (do not worry about them).

If you open up `index/hadoop/output/part-00000` you will see the output of the MR job, which is a word count job (just like last project!). **Please** read through `hadoop_pipeline`. It is crucial that you understand what is going on! It will help you debug problems with your full MR pipeline later on.

Important: Unfortunately, it is not uncommon for WSL to get stuck when running Hadoop. If this is the case, you may repeatedly try to kill the Hadoop processes and re run the jobs individually.

When experiencing this, your best option is likely to test via piping your input through the pipeline jobs directly. For example:

```
$ cat index/hadoop/input/* \
| index/hadoop/exec/inverted_index/map0.py \
| sort | index/hadoop/exec/inverted_index/reduce0.py
```

This would compute the first phase of your pipeline.

If you do not wish to pipe input/output directly and would prefer to use Hadoop itself on another platform, you are welcome to attempt setting up vagrant with a virtualbox virtual machine.
We have some instructions detailing this here. There are also some notes in the slides for Discussion 1.

Part 1: MapReduce Indexing

You will be building and using a MapReduce based indexer in order to process the large dataset for this project. You will be using Hadoop’s command line streaming interface that lets you write your indexer in the language of your choice instead of using Java (which is what Hadoop is written in). However, you are limited to using Python 3 so that the course staff can provide better support to students. **In addition, all of your mappers must output both a key and a value (not just a key). Moreover, keys and values BOTH must be non-empty. If you do not need a key or a value, common practice is to emit a “1”**

There is one key difference between the MapReduce discussed in class and the Hadoop streaming interface implementation: In the Java Interface (and in lecture) one instance of the reduce function was called for each intermediate key. In the streaming interface, one instance of the reduce function may receive multiple keys. **However, each reduce function will receive all the values for any key it receives as input.**

You will not actually run your program on hundreds of nodes; it is possible to run a MapReduce program on just one machine: your local one. However, a good MapReduce program that can run on a single node will run fine on clusters of any size. In principle, we could take your MapReduce program and use it to build an index on 100B web pages.

For this project you will create an inverted index for the documents in `index/hadoop/map_reduce_input_data` through a series of MapReduce jobs. This inverted index will contain the idf, term frequency, and document normalization factor as specified in the information retrieval lecture slides. The format of your inverted index must follow the format, with each data element separated by a space: "word -> idf -> doc_id -> number of occurrences in doc_id -> doc_id's normalization factor (before sqrt)"

Sample of the correctly formatted output for the inverted index can be found in `index/hadoop/sample.txt` . You can also use this sample output to check the accuracy of your inverted index.

For your reference, here are the definitions of term frequency and idf:

$$W_{ik} = tf_{ik} * \log(N / n_k)$$

- T_k = term k in document D_i
- tf_{ik} = freq of term T_k in doc D_i
- idf_k = inverse doc freq of term T_k in collection C
  $idf_k = \log(\frac{N}{n_k})$

- N = total # docs in collection C
- n_k = # docs in C that contain T_k

And here is the definition for normalization factor for a document:

$$\sqrt{\sum_{k=1}^{t} (tf_{ik})^2 [\log(N / n_k)]^2}$$

Part A: Splitter Script

Each document has three properties: `doc_id`, `doc_title`, and `doc_body`. Your mapper code will be receiving input via standard in and line-by-line. As a result, the input is newline separated and each document is represented by 3 lines: 1st is the `doc_id`, 2nd is the `doc_title`, and the 3rd is the `doc_body`. This will allow you to read the input in your mapper function easily.

Input for Hadoop programs is an input directory, not file. As you will notice, our input is in one large file called `map_reduce_input_data` but your code runs with multiple mappers, and each mapper gets one input file. Consequently, you must write a custom Python script to break the large `mapreduce_input_data` input file with over 3,000 documents (3 lines each) into smaller files and place these files into the `index/hadoop/input/` directory. You can name your smaller input files however you like, as long as they are under the `input/` directory. **Do not split the input file into one file per document; this will make your MapReduce pipeline run very slowly! We do not require a specific number of smaller input files, but ~30 is fine.**

Part B: Hadoop Pipeline Script

This script will execute several MR jobs. The first one will be responsible for counting the number of documents in the input data. Name the mapper and reducer for this map0.py and reduce0.py, respectively. The next N jobs will be a "pipeline", meaning that you will chain multiple jobs together such that the input of a job is the output of the previous job.

Part B.1: Job 0

The first MapReduce job that you will create counts the total number of documents. This should be run with 30 mapper workers and only **ONE** reducer worker. The mapper and reducer executables should be named `map0.py` and `reduce0.py`, respectively. The reducer should save the total number of documents in a file called `total_document_count.txt`. The only data in this file will be an integer representing the total document count. The `total_document_count.txt` file should be created in whichever directory the pipeline is executed from, not the `hadoop/mapreduce/` directory. So in this case, since we expect to run the pipeline as ./bin/hadoop_pipeline from $PROJECT_ROOT, we expect this text file to be created in $PROJECT_ROOT (hint: Python already does this if you open a file normally).

Part B.2 MapReduce Pipeline

You will be going from large datasets to an inverted index, which involves calculating quite a few values for each word. As a result, you will need to write several MapReduce jobs and run them in a pipeline to be able to generate the inverted index. The first MapReduce job in this pipeline will get input from `index/hadooop/input/` and write its output to a directory that you specify in `hadoop_pipeline`. All future jobs in this pipeline will use the output directory from the previous job as the input directory for the current job, until the inverted index is constructed. `hadoop_pipeline` shows how to pipe the output from one MapReduce job into the next one.

To test your MapReduce program, we recommend that you make a new test file, with only 10-20 of the documents from the original large file. Once you know that your program works with a single input file, try breaking the input into more files, and using more mappers.

Each of your MapReduce jobs in this pipeline should have 30 mappers and 30 reducers. However, your code should still work when the number of mappers/reducers is changed. You may only use maximum of 9 MapReduce jobs in this pipeline (but the inverted index can be produced in fewer). The first job in the pipeline (the document counter) must have mapper and reducer executables named `map0.py` , `reduce0.py` respectively, and the second job should be `map1.py` , `reduce1.py` , etc.

The format of your inverted index must follow the format, with each data element separated by a space: `word -> idf -> doc_id -> number of occurrences in doc_id -> doc_id's normalization factor (before sqrt)`

Sample of formatted output can be found in `index/hadoop/sample.txt` . The order of words in the inverted index does not matter. If a word appears in more than one doc it does not matter which doc_id comes first in the inverted index. Note that the log for idf is computed with base 10 and that some of your decimals may be slightly off due to rounding errors. In general, you should be within 5% of these sample output decimals.

When building the inverted index file, you should use a list of predefined stopwords to remove words that are so common that they do not add anything to search quality ("and", "the", etc). We have given you a list of stopwords to use in `stopwords.txt` . This file is found in the $PROJECT_ROOT directory. This is because your hadoop pipeline should be run from $PROJECT_ROOT through "./bin/hadoop_pipeline".

When creating your index you should treat capital and lowercase letters as the same (case insensitive). You should also only include alphanumeric words in your index and ignore any other symbols. If a word contains a symbol, simply remove it from the string. Do this with the following code snippet:

```python
import re
re.sub(r'[^a-zA-Z0-9]+', '', word)
```

You should remove non-alphanumeric characters before you remove stopwords. Your inverted index should include both `doc_title` and `doc_body` for each document.

To construct the inverted index file used by the index server, `cat` all the files in the output directory from the final MapReduce job, and put them into a new file (i.e. `cat index/hadoop/output/* > inverted_index.txt`). This inverted_index.txt should then be moved to your `index/index` so it can be used by your Flask App.

MapReduce Specifications (Important)

There are a few things you must ensure to get your mappers and reducers playing nicely with both hadoop and our autograder. Here is the list:

- Emit both keys **and** values at all stages
- Do not emit empty keys **or** values
- Please have a newline at the end of each of your mapper and reducer outputs at **every stage**
- Split key value pairs using a tab. It should be as such “key\tvalue”

Sample Input:

```
1
The Document: A
This document is about Mike Bostock. He made d3.js and he's really cool
2
The Document: B
Another flaw in the human character is that everybody wants to build and nobody wants to do m
3
Document C:
Originality is the fine art of remembering what you hear but forgetting where you heard it. -
```

Sample Output:

```
character 0.47712125471966244 2 1 1.593512841936855
maintenance 0.47712125471966244 2 1 1.593512841936855
mike 0.47712125471966244 1 1 1.138223458526325
kurt 0.47712125471966244 2 1 1.593512841936855
peter 0.47712125471966244 3 1 2.048802225347385
flaw 0.47712125471966244 2 1 1.593512841936855
heard 0.47712125471966244 3 1 2.048802225347385
cool 0.47712125471966244 1 1 1.138223458526325
remembering 0.47712125471966244 3 1 2.048802225347385
laurence 0.47712125471966244 3 1 2.048802225347385
d3js 0.47712125471966244 1 1 1.138223458526325
made 0.47712125471966244 1 1 1.138223458526325
build 0.47712125471966244 2 1 1.593512841936855
document 0.0 2 1 1.593512841936855 3 1 2.048802225347385 1 2 1.138223458526325
```

```
originality 0.47712125471966244 3 1 2.048802225347385
bostock 0.47712125471966244 1 1 1.138223458526325
forgetting 0.47712125471966244 3 1 2.048802225347385
hear 0.47712125471966244 3 1 2.048802225347385
art 0.47712125471966244 3 1 2.048802225347385
human 0.47712125471966244 2 1 1.593512841936855
fine 0.47712125471966244 3 1 2.048802225347385
vonnegut 0.47712125471966244 2 1 1.593512841936855
```

Part 2: Index Server

The index server is a separate service from the search interface that handles search queries and returns a list of relevant results. Your index server is a RESTful API that returns search results in JSON format that the search server processes to display results to the user.

Directory Structure

In this project since you have two servers, the index server and the search server, you will need separate directories for each of your server applications. Your index server code is going to be inside the directory `index/index` and your `setup.py` is going to be inside the directory `index`. This is to allow you to use python packages with two different servers in the same $PROJECT_ROOT directory. Note that the `setup.py` is always in a directory above the directory in which your application's code is. Below is an example of what your final index directory structure should look like.

```
├── index
│   ├── hadoop
│   │   ├── exec
│   │   │   └── inverted_index
│   │   │       ├── map0.py
│   │   │       ├── mapX.py
│   │   │       ├── reduce0.py
│   │   │       ├── reduceX.py
│   │   ├── hadoop-streaming-2.7.2.jar
│   │   ├── mapreduce_input_data
│   │   └── split_file.py
│   ├── index
│   │   ├── *.py
│   │   ├── api
│   │   │   └── *.py
│   │   ├── inverted_index.txt
│   │   ├── pagerank.out
│   │   └── stopwords.txt
│   └── setup.py
```

Your `index` and `index/api` directories need to be python packages.

Part A: PageRank Integration

In lecture, you learned about PageRank, an algorithm used to determine the relative importance of a website based on the sites that link to that site, and the links to other sites that appear on that website. Sites that are more important will have a higher PageRank score, so they should be returned closer to the top of the search results.

In this project, you are given a set of pages and their PageRank scores in `pagerank.out`, so you do not need to implement PageRank. However, it is still important to understand how the PageRank algorithm works!

Your search engine will rank documents based on both the query-dependent tf-idf score, as well as the query-independent PageRank score. The formula for the score of a query q on a single document d should be:

$$Score(q, d, w) = w * PageRank(d) + (1 - w) * tfIdf(q, d)$$

where **w** is a decimal between 0 and 1, inclusive. The value **w** will be a URL parameter. The **w** parameter represents how much weight we want to give the document's PageRank versus its cosine similarity with the query. This is a completely different variable from the "w_{ik}" given by the formula in Part 1. The final score contains two parts, one from pagerank and the other from a tf-idf based cosine similarity score. The **PageRank(d)** is the pagerank score of document **d**, and the **tfIdf(q, d)** is the cosine similarity between the query and the document tf-idf weight vectors. Each weight in a tf-idf weight vector is calculate via the formula in Part 1 for "w_{ik}". Treat query **q** as a simple AND, non-phrase query(that is, assume the words in a query do not have to appear consecutively and in-sequence).

Integrating PageRank scores will require creating a second index, which maps each `doc_id` to its corresponding precomputed PageRank score, which is given to you in `pagerank.out`. You can do this where you read the inverted index. This index should be accessed at query time by your index server.

If you still have some confusion with these calculations, please refer to Appendix A

Part B: Returning Hits

When the Index server is run, it will load the inverted index file, pagerank file, and stopwords file into memory and wait for queries. **It should only load the inverted index and pagerank into memory once, when the index server is first started.** Every time you send an appropriate request to the index server, it will process the user's query and use the inverted index loaded into memory to return a list of all of the "hit" `doc_ids`. A hit will be a document that is similar to the user's query. When finding similar documents, only include documents that have every word from the query in the document. The index server should not load the inverted index or pagerank into memory every time it receives a query!

Search Endpoint

Route: `http://{host}:{port}?w=<w>&q=<query>`

Your index server should only have one endpoint, which receives the pagerank weight and query as URL parameters **w** and **q**, and returns the search results, including the relevance score for each result (calculated according to the previous section). For example, the endpoint `?w=0.3&q=michigan%20wolverine` would correspond to a pagerank weight of 0.3, and a query "michigan wolverine".

Your index server should return a JSON object in the following format:

```
{
  "hits": [
    {
      "docid": 868657,
      "score": 0.071872572435374
    }
  ]
}
```

The documents in the hits array must be sorted in order of relevance score, with the most relevant documents at the beginning, and the least relevant documents at the end. If multiple documents have the same relevance score, sort them in order of `doc_id` (with the smallest `doc_id` first).

When you get a user's query make sure you remove all stopwords. Since we removed them from our inverted index, we need to remove them from the query. Also clean the user's query of any nonalphanumeric characters as you did with the inverted index.

Make sure that `stopwords.txt` is in the same directory as your index server executable.

Running the Index Server

Install you index server app by running the command below in your project root directory:

```
pip install -e index
```

To run the index server, you will use environment variables to put the development server in debug mode, specify the name of your app's python module and locate an additional config file. (These commands assume you are using the bash shell.)

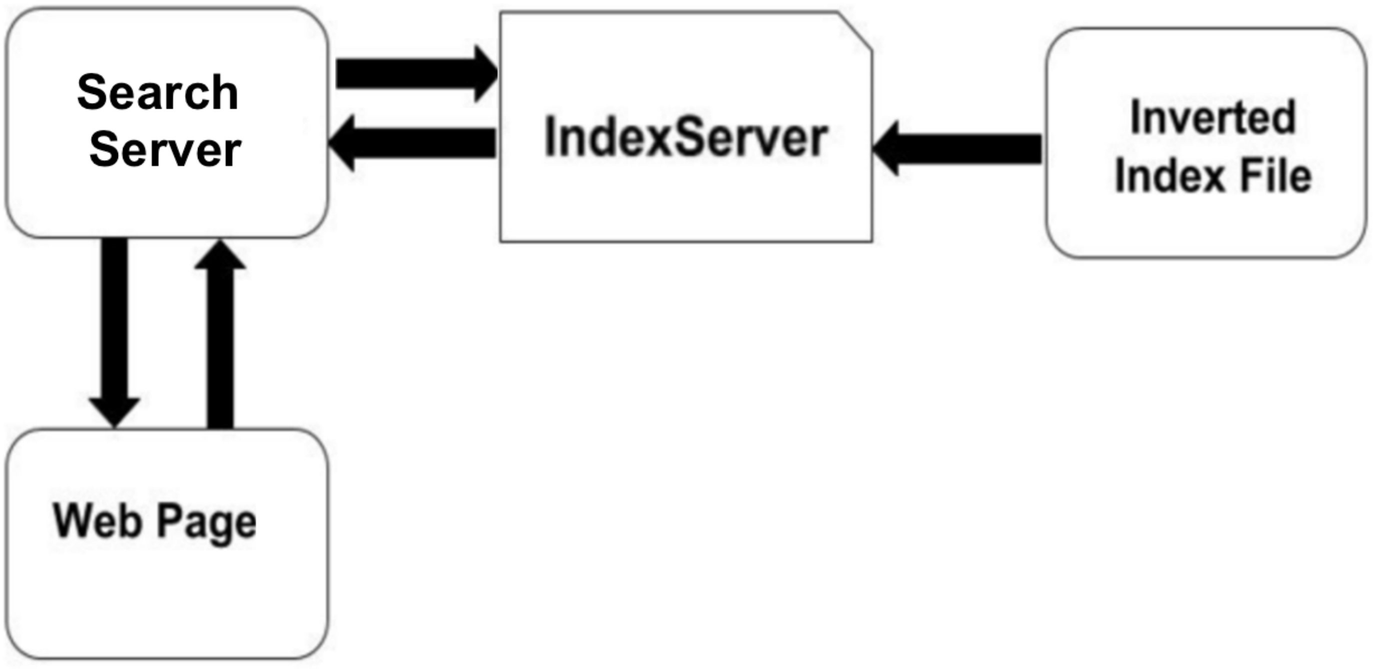
```
$ export FLASK_DEBUG=True
$ export FLASK_APP=index
$ export INDEX_SETTINGS=config.py
$ flask run --host 0.0.0.0 --port 8001
 * Serving Flask app "index"
 * Forcing debug mode on
 * Running on http://127.0.0.1:8001/ (Press CTRL+C to quit)
```

Deliverables

- Write an index server which uses the inverted index and pagerank files to return an appropriately formatted JSON object with a vector of documents that are similar to the search query.

Part 3: Search Interface

The third component of the project is a React driven interface for the search engine. The search interface app will provide a GUI for the user to enter a query and specify a Pagerank weight, and will then send a request to your index server. When it receives the search results from the index server, it will display them on the webpage.



Directory Structure

As mentioned earlier, in this project since you have two separate servers, the index server and the search server, you will need a separate directory for you search server. Your search interface server code is going to be inside the directory `search/search` and your `setup.py` is going to be inside the directory `search`. Your search server is supposed to be a python package. Below is an example of what your final search directory structure should look like.

```
├── search
│   ├── package.json
│   ├── package-lock.json
│   ├── node_modules
│   │   └── */
│   ├── search
│   │   ├── api
│   │   │   └── *.py
│   │   ├── js
│   │   │   └── *.jsx
│   │   ├── server_config.py
│   │   ├── sql
│   │   │   └── wikipedia.sql
│   │   ├── static
│   │   │   └── js
│   │   │       └── bundle.js
│   │   ├── templates
│   │   │   └── *.html
│   │   ├── var
│   │   │   └── wikipedia.sqlite3
│   │   └── views
│   │       └── *.py
│   ├── setup.py
│   └── webpack.config.js
```

Your `search`, `views`, and `api` directories need to be python packages.

Part A: Database

You will need to create a new database for project 5, with a table called Documents as follows:

- `docid`: `INT` and `PRIMARY KEY`
- `title`: `VARCHAR` with max length 100
- `categories`: `VARCHAR` with max length 5000
- `image`: `VARCHAR` with max length 200
- `summary`: `VARCHAR` with max length 5000

The SQL to create this table and load the necessary data into it is provided in `search\search\sql\wikipedia.sql` inside the starter files.

Part B: GUI

The following is a list of the specifications for your search GUI

Search interface

Route: `http://{host}:{port}/`

You will be making a simple search interface for your database of Wikipedia articles which allows users to input a **query** and a **w** value, and view a ranked list of relevant docs. Users can enter their **query** into a text input box (`<input type="text">`), and specify the **w** value using a slider. The query is executed when a user presses a (`<input type="submit">`). You can assume anyone can use your search engine; you do not need to worry about access rights or sessions like past projects. Your engine should receive a simple AND, non-phrase query (that is, assume the words in a query do not have to appear consecutively and in-sequence), and return the ranked list of docs. Your search results will be titles of Wikipedia documents, displayed exactly as they appear in the database. Search results will show **at most 10 documents**

Note: any other content should appear *below* this interface

This page must include a slider (`<input type="range">`) that will set the **w** GET query parameter in the URL, and will be a decimal value between 0-1, inclusive. You should set the range slider's step value to be 0.01.

When the user clicks the submit button of the search GUI, you should fetch the appropriate titles from the index server. For each title returned, display the title in a p tag with a class="doc_title" (`<p className="doc_title">`) as well as a button to display the summary for that title with a value="Show Summary"(`<input type="submit" value="Show Summary">`)

If there are no search results, no action is required

Summary

Upon clicking the 'show summary' button for a title, the list of results sould be *replaced* with a summary of the document. You may simply place the summary data within a p tag with a class="doc_summary"(`<p className="doc_summary">`) In addition to having this summary, this page will also have a "Similar Documents" section, which will show **at most 10 documents** using w=0.15, and the current document's title as the search query, with the underscores replaced with spaces. For example, if the title of the document is "Saint_Petersburg", the search query should be "Saint Petersburg". For each related document title, render a p tag with class="doc_title" (`<p className="doc_title">`)

Running the Search Server

Install you search server app by running the command below in your project root directory:

```
pip install -e search
```

To run the search server, you will use environment variables to put the development server in debug mode, specify the name of your app's python module and locate an additional config file. (These commands assume you are using the bash shell.)

```
$ export FLASK_DEBUG=True
$ export FLASK_APP=search
$ export SEARCH_SETTINGS=config.py
$ flask run --host 0.0.0.0 --port 8000
 * Serving Flask app "search"
 * Forcing debug mode on
 * Running on http://127.0.0.1:8000/ (Press CTRL+C to quit)
```

In order, to be able to search on the search server, you must have your index server up and running, since it is the response of the index server that the search server uses to display the search results.

Deliverables

- Write a search server, which is a GUI application to allow users to search the Wikipedia database. Your search server requests search query results from the index server and uses the JSON response to display the search results to the user.

Shell Scripts to Launch both Servers

You will be responsible for writing shell scripts to launch both the index server and the search server. Hint: these will be somewhat similar to last project's bin/mapreduce script. The name of these scripts must be "bin/index" and "bin/search"

Example: start search.

```
starting search server ...
+ export FLASK_APP=search
+ export SEARCH_SETTINGS=config.py
+ flask run --host 0.0.0.0 --port 8000 &> /dev/null &
```

Example: stop search. (note: the pid that gets killed won't always be the same)

```
stopping search server ...
+ kill -9 48796
```

Example: restart search. (note: the pid that gets killed won't always be the same)

```
stopping search server ...
+ kill -9 48769
starting search server ...
+ export FLASK_APP=search
+ export SEARCH_SETTINGS=config.py
+ flask run --host 0.0.0.0 --port 8000 &> /dev/null &
```

Example: start search when something is already running on port 8000

```
Error: a process is already using port 8000
```

Example: start index.

```
starting index server ...
+ export FLASK_APP=index
+ flask run --host 0.0.0.0 --port 8001 &> /dev/null &
```

Example: stop index. (note: the pid that gets killed won't always be the same)

```
stopping index server ...
+ kill -9 48711
```

Example: restart index. (note: the pid that gets killed won't always be the same)

```
stopping index server ...
+ kill -9 48692
starting index server ...
+ export FLASK_APP=index
+ flask run --host 0.0.0.0 --port 8001 &> /dev/null &
```

Example: start index when something is already running on port 8001

```
Error: a process is already using port 8001
```

Hint: `lsof -i:<port_num>` will display the name of the process that is running on `port_num` . You can use this to test if something is "occupying" `port_num` . **This won't work in WSL.** If you are using WSL, the following command will give you the process id of the index server: `ps aux | grep 'flask` `run \-\-host 0.0.0.0 \-\-port 8001' | awk '{print $2}'` . Change 8001 to 8000 in that command to retrieve the process id of the search server.

`flask run --host 0.0.0.0 --port 8000 &> /dev/null &` will start up your flask server in the background. The `&> /dev/null` will prevent your Flask app from outputting text to the terminal.

DB Management Script

You will also be expected to write a database script that supports options: create, destroy and reset.

Example: bin/db create

```
+ sqlite3 search/search/var/wikipedia.sqlite3 < search/search/sql/wikipedia.sql
```

Example: bin/db create (db already exists)

```
Error: database already exists
```

Example: bin/db destroy

```
+ rm -rf search/search/var/wikipedia.sqlite3
```

Example: bin/db reset

```
+ rm -rf search/search/var/wikipedia.sqlite3
+ sqlite3 search/search/var/wikipedia.sqlite3 < search/search/sql/wikipedia.sql
```

Deploying to AWS

Please refer to the AWS instructions linked in Project 2. If you were successful in deploying Project 2, you will need to log into your instance via ssh and follow the instructions listed under “Deploy Instructions (Every Project).” **Note** similar to Project 3, you used npm and webpack to run Project 5, so we need to set up our virtual environment accordingly. Also be aware that we need to run two servers this time around (index server and search server) **In addition, we need to do one additional step, which is to obfuscate our Javascript code.** This is so that others cannot download our original, readable Javascript source code. See here:

```
$ cd project_repo_dir_name/
$ python3 -m venv env
$ source env/bin/activate
$ pip install -e search/
$ pip install -e index/
$ pip install gunicorn
$ pip install nodeenv
$ nodeenv --python-virtualenv
$ source env/bin/activate  # again, after installing node
$ cd search/
$ npm install .
$ node_modules/.bin/webpack
$ npm install javascript-obfuscator # install an obfuscator package
$ node_modules/.bin/javascript-obfuscator search/static/js/bundle.js  #obfuscate our webpack
$ mv search/static/js/bundle-obfuscated.js search/static/js/bundle.js #replace with the obfus
$ cd ../
$ bin/db create
```

Note that there will be two gunicorn commands this time, and they are slightly different:

```
$ gunicorn -b localhost:8000 -w 2 -D search:app
$ gunicorn -b localhost:8001 -w 2 -D index:app
```

The above instructions, regarding your virtual environment setup and code ofuscation, are the only difference with what is provided in the AWS doc. **All of the instructions regarding gunicorn should still be followed!**

Code Style

As in previous projects, all your Python code for the Flask servers is expected to be `pycodestyle`, `pydocstyle`, and `pylint` compliant. You don't need to lint the mapper/reducer executables (although it may be a good idea to do so still!)

Also as usual, you may not use any external dependencies aside from what is provided in the setup.py's.

Submission

Submission for this project will be slightly different than previous projects, due to the structural complexity of this project. We have provided a `bin/submit` script which you can run to execute the tar command for this project. Open the submit script up to inspect it. You will notice it is similar to previous projects' tar commands, but has a bunch of "exclude" flags. This is to prevent you from submitting extremely large files unecessarily to the AG. There are three variables ($MR_INPUT_DATA,

$MR_INPUT_FOLDER, $MR_OUTPUT_FOLDER). Make sure these variables are appropriately set (the script discusses them in greater details in comments).

If you are on OSX please add the disable-copyfile flag as usual. It should look something like:

```
tar -czvf submit.tar.gz --exclude="something" ... --exclude="something_else"
--disable-copyfile bin index search
```

Feel free to modify this script to exclude other unnecessary resources you should not be submitting (e.g. any extra text files you've created yourself etc.).

Deployment Submission

To generate the deployment submission files please follow the commands below. The submit script in `bin/submit` will take care of including these files in the tarball.

Run these commands in your $PROJECT_ROOT directory after deploying your website. **Reminder: "hostname" refers to your DNS hostname, found under "Public DNS (IPv4)" in the AWS console. Not your IP!**

```
$ curl -v "http://<hostname>/" 2> deployed_search.log
$ curl -v "http://<hostname>/" > deployed_search.html
```

Here is a snippet of the location of these files in the $PROJECT_ROOT:

```
├── bin
│   ├── hadoop_pipeline
│   ├── index
│   ├── search
│   └── submit
├── deployed_index.json
├── deployed_index.log
├── deployed_search.html
├── deployed_search.log
├── index
...
...
...
```

Rubric

This is an approximate rubric.

Deliverable	Value
Pipeline	35%
Index Server	35%
Search Interface	25%
Code Style and Scripts	5%

Private tests will account for 17% in final grade calculation.

Important Info

- `bin/submit` to create tarball.
  - Do not submit any extraneous files
  - Add extra exclude flags if need be
  - Add disable copyfile flag if on OSX
- Make sure that your index server reads `stopwords.txt`, `pagerank.out` from the `index/index` directory (please use absolute paths for this - hint: `os.path.dirname` and `__file__` are your friends!)
- Your mappers and reducers should **not** be using absolute paths for their files. They should access `stopwords.txt` simply by filename like so:
  - `f = open("stopwords.txt", "r")`

FAQ

API Request URLs

When you are attempting to send a request to **search interface API** from your **ReactJS search interface frontend**, do not include the `host:port` in the url you are attempting to fetch.

If you want to send a request to your **index server API** from your **search interface API**, you will want to include the following `host:port` pair in the url you are attempting to send a request to: `http://localhost:8001`

OSX Java Home Error

1. Find `hadoop-env.sh`. You location might be different.

```
$ find / -name hadoop-env.sh
/usr/local/Cellar/hadoop/2.8.1/libexec/etc/hadoop/hadoop-env.sh
```

2. Uncomment and edit the `export JAVA_HOME` line in `hadoop-env.sh` :

   ```
   # The java implementation to use.
   export JAVA_HOME="$(/usr/libexec/java_home)"
   ```

Appendix A: TFIDF Calculation Walkthrough

You have a query, let's make a query vector for that

$$\bar{q} = [tf_{q1} * idf_1, tf_{q2} * idf_2, \ldots]$$

You have a document, let's make a document vector for that

$$\bar{d} = [tf_{d1} * idf_1, tf_{d2} * idf_2, \ldots]$$

Let's define a norm factor for our document

$$norm_d = \sqrt{\sum_k (tf_k)^2 (idf_k)^2}$$

Let's define a norm factor for our query

$$norm_q = \sqrt{\sum_k (tf_k)^2 (idf_k)^2}$$

Note: that the IDFs for both norm factors come from the inverted index. The norm factor for the document can be lifted from your inverted index (but you have sqrt it), where as the query norm factor needs to be computed on the fly.

If you have defined your query and document vectors this way, then the final score will be

$$tf - idfscore(\bar{q}, \bar{d}) = \frac{\bar{q} * \bar{d}}{norm_q * norm_d}$$